NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

GENERALIZED BOOLEAN FUNCTIONS AS COMBINERS

by

Oliver Di Nallo

June 2017

Thesis Advisor:	Pantelimon Stănică
Second Reader:	Thor Martinsen

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2017	Master's Thesis 07-05-2016 to 06-16-2017

4. TITLE AND SUBTITLE

GENERALIZED BOOLEAN FUNCTIONS AS COMBINERS

5. FUNDING NUMBERS

6. AUTHOR(S)

Oliver Di Nallo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

In this digital age, cryptography has formed the backbone of many computer functions. Cryptography drives online commerce and allows privileged information safe transit between two parties as well as many other critical internet uses. The presence of a strong pseudo-random number generator (PRNG) is an absolute requirement in modern cryptography. All modern ciphers draw their strength from having this strong generator. There are currently many ways to generate a secure PRNG. Most current PRNGs generate their stream as a sequence of bits. As a result, most tests performed to ensure randomness are made for binary streams. This thesis introduces a way to generate an integer random number stream using generalized Boolean functions. Additionally, this thesis discusses how to test an integer stream using binary tests. Data from this thesis suggests that high levels of complexity can be obtained using simple quadratic (or other higher degree) generalized combiners. Additionally, our data discusses the ability to generate sequences with high degrees of randomness using a variety of combiner choices for the generalized Boolean function.

14. SUBJECT TERMS	15. NUMBER OF PAGES
pseudo-random number generator, generalized Boolean function	65
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

GENERALIZED BOOLEAN FUNCTIONS AS COMBINERS

Oliver Di Nallo
Second Lieutenant, United States Army
B.S., United States Military Academy, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2017

Approved by: Pantelimon Stănică
Thesis Advisor

Thor Martinsen
Second Reader

Craig Rasmussen
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In this digital age, cryptography has formed the backbone of many computer functions. Cryptography drives online commerce and allows privileged information safe transit between two parties as well as many other critical internet uses. The presence of a strong pseudo-random number generator (PRNG) is an absolute requirement in modern cryptography. All modern ciphers draw their strength from having this strong generator. There are currently many ways to generate a secure PRNG. Most current PRNGs generate their stream as a sequence of bits. As a result, most tests performed to ensure randomness are made for binary streams. This thesis introduces a way to generate an integer random number stream using generalized Boolean functions. Additionally, this thesis discusses how to test an integer stream using binary tests. Data from this thesis suggests that high levels of complexity can be obtained using simple quadratic (or other higher degree) generalized combiners. Additionally, our data discusses the ability to generate sequences with high degrees of randomness using a variety of combiner choices for the generalized Boolean function.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Cryptography has become embedded into the very fabric of the internet. Almost every transaction or exchange of data now involves some sort of encryption. Modern encryption standards are thought to be incredibly secure; however, that claim is invalided if there is not a strong source to randomness for the encryption methods to draw their strength from. Generating truly random numbers is possible, but impractical. Instead, pseudo-random number generators(PRNG) provide a way of quickly and efficiently generating streams that appear very random.

One way to generate a PRNG is to use the output of Linear Feedback Shift Registers (LFSRs). Additionally, a combining Boolean function is used with an LFSR to destroy the linearity of the LFSRs output stream. Using combiners with LFSR is commonplace. This thesis introduces a method of generating pseudo-random number sequences using a generalized Boolean function. Not much is known about the properties of using generalized Boolean functions as combiners. One of the effects of using a generalized Boolean function is that the output stream can consist of integers, rather than bits if a Boolean combiner is used.

Data from our thesis shows that a significant increase of complexity can be achieved by a generalized Boolean function output when compared to the output of a Boolean combiner. Additionally, our data shows that high levels of randomness can be obtained using this method.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank the United States Army Cyber Command for allowing me to take a year to come to the Naval Postgraduate School to study applied mathematics. I would also like to thank my thesis adviser, Dr. Pante Stanica, for mentoring me and providing such wonderful insight into this thesis. Additionally, I would like to thank CDR Thor Martinsen for the guidance that he also provided during this process.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

To claim good modern cryptography without a proper source of randomness is an impossible task. Randomness forms the background of almost all modern forms of cryptography through its prevalent use in public/private key generation, session key generation, and nonces among many other uses. While it is so important, finding a true random number generator efficiently is next to impossible; so, certain compromises to provable security must be made. Such a random number generator is called pseudo-random number generator (PRNG) and is the subject of this thesis.

To classify as a PRNG, two main criteria need to be met to maintain confidence in its randomness. The first is that the stream is statistically random, and the second is that stream is unpredictable. Testing the randomness of streams to make sure they satisfy those two properties and more is possible through the suite of tests published by the National Institute of Standards and Technology (NIST). Many methods exist for generating a pseudo-random stream of numbers. Some examples include the Blum-Blum-Shub generator, the Linear Congruential Generator, and Linear Feedback Shift Registers (LFSR) combined with Boolean functions. This thesis demonstrates a new method of generating a random stream through the output of generalized Boolean functions, and shows that this output passes a majority of the randomness tests in addition to showing a significant increase in complexity.

1.1 Background

Cryptography has existed for thousands of years. In its earliest form, simple ciphers were used to mask battle plans or carry hidden messages between spies. The Caesar cipher, purported to be used by Julius Caesar, is the most famous of these. The so called 'Caesar cipher' was just a simple shift cipher that shifted every letter in the message three steps in the alphabet. For example the phrase

$$weattackatdawn$$

put into a Caesar cipher that shifts every character 3 letters would produce the ciphertext

$$zhdwwdfndwgdzq.$$

Since then cryptography has become significantly more complex. Today's ciphers are based on rigorous mathematics, and are thought to be extremely secure.

1.1.1 Ciphers

Of all of today's ciphers, the only cipher that has perfect, provable security is the one-time pad cipher. The one-time pad cipher takes a stream of random bits as its key, then performs the exclusive OR (XOR) operation on a bit of the key and a bit of the message for the entire length of the message. For example, if the message is 01011 and the key is 10110, the ciphertext will be 11101. As long as the key was generated from a random source, and the key is not reused on any part of the message, it is impossible to crack this code. However, this code is very cumbersome to use, since the key needs to be as long as the message, can never be reused, and must be in the possession of both parties. Modern encryption standards make slight sacrifices to provable security in order to provide ease of use, while still maintaining acceptable levels of security.

One of the most widespread ciphers in use today is the Advanced Encryption Standard (AES). AES is an example of a block cipher, where the plaintext message is processed in chunks. AES is considered very secure, and it is the U.S. government's recommendation for encrypting classified documents as reported in [1]. AES, and block ciphers in general, are considered symmetric ciphers. A symmetric cipher is a cipher where the same key is used to encrypt and decrypt the communication. In addition to a key, a good source of randomness is required in order to make the encryption secure.

1.1.2 Randomness

There are many sources of true randomness that could be used in a cipher. For example, the atmospheric noise generated as a result of solar radiation is considered a random source. The website, www.random.org, publishes random numbers generated through listening to this source of noise [2]. The Handbook of Applied Cryptography [3] lists many other sources such as the radioactive decay of atoms and the vibrations resulting from the turbulent flow

of a hard drive platter spinning through air. While there are several ways to generate true random numbers, none of these are practical for widespread use in cryptography.

PRNGs are used to provide a secure, yet efficient, way of creating seemingly random number sequences. PRNGs are by definition deterministic since they take a seed that will always generate the same output. The strength in a PRNG comes from the requirement that they have such complexity that it is computationally impossible to recover the generating function used to create the stream. Many types of PRNGs exist, but this thesis focuses entirely on the PRNGs that use LFSRs and combiners. As mentioned in [4], LFSRs are used extensively in stream ciphers to generate the random bits required. LFSRs are also very popular for use in embedded systems because they can be easily coded in hardware. However, LFSRs are, by themselves, not secure by. They have high degrees of linearity, and Boolean functions called combiners are required to destroy their linearity in order to make them more secure.

1.2 Motivation

Current PRNGs that utilize LFSRs and combiners are cryptographically secure and in common use. Much is known about their randomness and complexity and they have been the subject of academic research for at least the past 50 years. Since LFSRs are linear by nature, calculating their actual complexity can be efficiently done using the Berlekamp-Massey algorithm. By design, these algorithms produce a seemingly random bit stream that is composed of an alphabet of 2 bits, 0 and 1. What a generalized Boolean function provides is the ability to produce a stream of integers in $\mathbb{Z}_n$ (integers modulo n) in a single computation cycle. With regular combiners, only 1 bit at a time is possible per computation round. In order to get enough bits to form an integer in $\mathbb{Z}_n$, multiple rounds need to be performed. With a generalized Boolean function, obtaining an integer only requires one round, so significant decreases in computation time can be achieved. Additionally, adding another combining step has the potential to significantly increase the complexity of the sequence, which is a result that this thesis shows.

Having a PRNG that outputs a stream of integers has several uses. One application could be its use as lookup indexes for matrices where the lookup index needs to be randomly generated. For example it could be used as the lookup function for the S-boxes of AES or

DES. Additionally, it could see use in Quantum computing where computations occurs in $\mathbb{Z}_4$ instead of $\mathbb{Z}_2$.

Not much prior work has been done investigating the combining properties of generalized Boolean functions, so the work this thesis presents is novel. Specific areas on interest include the complexity of these streams and whether or not they are sufficiently random for use in cryptographic applications.

CHAPTER 2:
Generalized Boolean Functions

This chapter provides the mathematical background of how the pseudo-random sequences of this thesis were generated. Additionally, it discusses what the linear complexity of a sequence is and how it can be calculated. The chapter also provides a description of LFSRs, Boolean combiner functions, and generalized Boolean functions and it also considers how an output stream from a generalized Boolean function can be used as a pseudo-random number generator.

Of the available choices for PRNGs, LFSRs are very popular for many reasons. LFSRs are efficient to implement in hardware and have easily calculable properties. However, additional levels of operations are required in order to make it cryptographically secure. Boolean functions, which can be used as combiners, are chosen to increase the complexity of the LFSR through a number of ways. A generalized Boolean function takes the process a step further by using the output bits from multiple combiners to create a stream of integers rather than bits.

2.1 Linear Feedback Shift Registers

An LFSR can be defined as a series of stages, or registers, where each stage has an input bit, an output bit, and a feedback function that acts upon it. The movements of bits are governed by a clock, with one step in the clock cycle advancing all the bits in the LFSR forward one position. The output stream is composed of the content of the output bit of the first stage of the LFSR at every clock cycle [3]. The content from the last stage is fed into the feedback function. The feedback function operates only on certain stages, called taps, and XORs the contents of the bit from the last stage with the contents of the taps sequentially to generate the input bit of first stage of the LFSR. This can be represented using Definition 2.1.1 which is taken from [3].

Definition 2.1.1 *Given the initial state of an LFSR as* $[s_{L-1}, ..., s_1, s_0]$*, the output sequence*

$s = s_0, s_1, s_2, \ldots$ *can be determined by the following equation*

$$s_j = (c_1 s_{j-1} + c_2 s_{j-2} + \cdots + c_L s_{j-L}) \mod 2 \quad for \quad j \geq L.$$

A feedback function can be represented either as a recurrence relation or as a polynomial. The polynomial has a general form of

$$P(x) = 1 + \sum_{i=1}^{L} c_i x^i,$$

where c_i are the taps and L is the order of the LFSR. For example, the following two LFSRs are the same LFSR represented in two different ways. The following equation is an LFSR in recurrence form

$$x_{i+4} = x_i \oplus x_{i+1}. \tag{2.1}$$

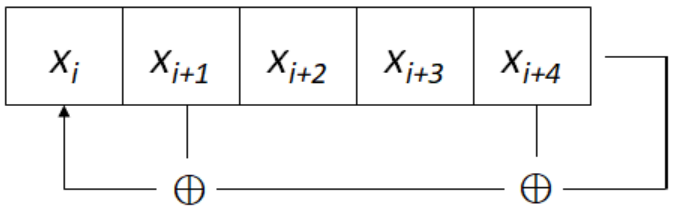
With 2.1, it is obvious how the LFSR is constructed, with the next bit relying on the generating function of the previous bits. The next equation is the same LFSR in its polynomial form

$$1 + x + x^4. \tag{2.2}$$

The polynomial form is the more common way of representing an LFSR. The highest power of the polynomial determines the order of the LFSR.

Figure 2.1 shows this LFSR graphically with the stages and taps to provide a visual way of representing the recurrence relation and the polynomial.

Figure 2.1. The LFSR from Equation 2.2 Graphically Depicted.



In order to work, an LFSR needs to be seeded with initial values. It is important that the

initial seed be of a good random source in order to make the output of the LFSR hard to predict. Once seeded, the number of iterations required to regenerate the seed of the LFSR is called the period, or length, of the LFSR. In Example 2.1.1, the LFSR generated from Equation 2.2 will be examined to show how an LFSR works in practice.

Example 2.1.1 *The values of* $1 + x + x^4$

For this example, the LFSR has a seed of 11010*. Table 2.1 will show the values of LFSR at every step.*

Table 2.1. The Stream Generated in Example 2.1.1

Step	x_i	x_{i+1}	x_{i+2}	x_{i+3}	x_{i+4}
0	1	1	0	1	0
1	0	1	1	0	1
2	0	0	1	1	0
3	0	0	0	1	1
4	1	0	0	0	1
5	0	1	0	0	0
6	1	0	1	0	0
7	1	1	0	1	0

The output stream from this LFSR is 0001011*, which is the first bit from every cycle.*

A non-singular LFSR is an LFSR where the degree of its feedback polynomial is equal to the order of the LFSR. This result is interpreted in [4] as any sequence by a non-singular LFSR of length L that is periodic and whose period does not exceed $q^L - 1$.

A polynomial is considered primitive if it is irreducible over $\mathbb{Z}_2$. A more general definition that comes from [3] is Definition 2.1.2.

Definition 2.1.2 *An irreducible polynomial* $f(x) \in \mathbb{Z}_p[x]$ *of degree* m *is called a primitive polynomial if a root of* f *is a generator of* $\mathbb{F}^*_{p^m}$*, the multiplicative group of all the non-zero elements in* $F_{p^m} = \mathbb{Z}_p[x]/f(x)$*.*

In less formal terms, the polynomial is primitive if any of its roots generate the multiplicative group $\mathbb{F}^*_{p^m}$ and is irreducible. There exists much research into primitive polynomials, and extensive tables have been published that list all primitive polynomials up to a certain degree. We used primitive polynomials for this thesis, and the polynomial in Equation 2.2 is primitive. One of the consequences of having a primitive polynomial is that the period of the non-singular LFSR will be maximal for all non-trivial seeds.

LFSRs have several properties that make them ideal for use in cryptography. The first is that they are very easy to implement in hardware. An LFSR only requires bit shifts and logical operations, so all instructions can be handled at the hardware level and can be performed very efficiently. Additionally, when chosen with a primitive polynomial as the feedback function, LFSRs can have a very large period, which hardens them against certain cryptographic attacks. LFSRs with a maximal period will have good statistical properties such that "the distribution of patterns having fixed length of at most L is almost uniform," which is a useful property for cryptography [3]. LFSRs, however, have some drawbacks due to the fact that they are very linear in nature and can be easily regenerated using the Berlekamp-Massey algorithm. This can be avoided through the use of a Boolean function to greatly increase the linear complexity.

2.2 Combiners

In general, a combiner can create non-linear output from LFSRs in multiple ways. The first is to take the outputs of several LFSRs and feed it into one combining Boolean function. The second way is to use a non-linear filter generator that takes the output of one LFSR at multiple stages and applies a filtering function to the bits. The third way is to use a clock-controlled generator that uses two LFSRs. The first LFSR provides the output bits, and the second LFSR instructs which of those bits to take for the stream. The first method of combining LFSRs is used in this thesis.

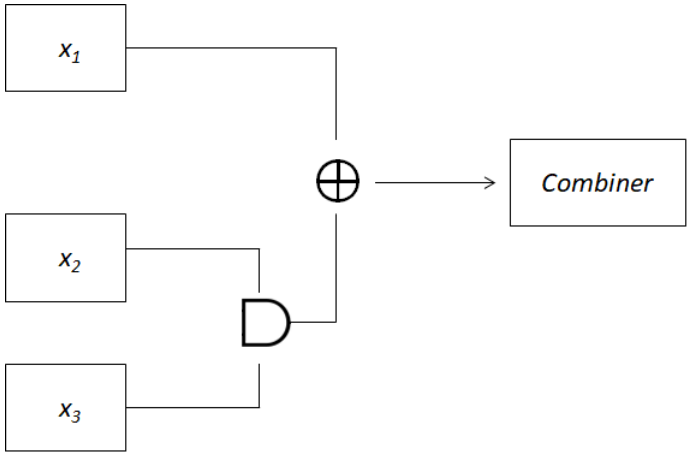
A combining Boolean function works by taking the output of several LFSRs, then applies a Boolean function to those bits to create a single output bit. Boolean functions are typically represented in their algebraic normal form. An algebraic normal form is written as a function of the sum of products of the input bits. A combiner operates in $\mathbb{Z}_2$ so all the operations can be reduced to logical ANDs and XORs. The order of a Boolean combining function

can be determined by finding the order of the highest term in the function. Equation 2.3 is an example of a combiner of order 2 its in algebraic normal form.

$$f(x_1, x_2, x_3) = x_1 \oplus x_2 x_3 \tag{2.3}$$

Figure 2.2 is Equation 2.3 depicted graphically.

Figure 2.2. A Combiner of the Form $f(x_1, x_2, x_3) = x_1 \oplus x_2 x_3$



So in Equation 2.3, x_1 would come from the current output bit of the first LFSR, x_2 from the second LFSR and x_3 from the third LFSR.

There are several things to take into account when selecting a Boolean function. One thing to consider is the degree of the involved function. For example, an affine function would be the Boolean function,

$$f(x) = x_1 \oplus x_2,$$

while a quadratic function would be of the form

$$f(x) = x_1 x_2 \oplus x_2.$$

One of the main purposes of using combiners is to destroy the linearity of an LFSR. To achieve this end, higher degree Boolean functions are always chosen since they tend to have significantly higher complexity than affine Boolean functions.

Selection of the LFSRs used for input of a combiner needs to be done carefully in order to ensure the combiner has maximal period and is cryptographically secure. One of the main requirements is that the order of the LFSRs be pairwise relatively prime. Another requirement is that the LFSRs be generated by primitive polynomials. With both of these requirements met, the combiner will have maximal period (due to of primitiveness) and maximal linear complexity (due to the relatively prime LFSRs). The linear complexity of the combiner can be calculated by substituting in the orders of the respective LFSRs into the polynomial representation of the combiner.

From [4], the linear complexity of the sequence $u + v$ is

$$L(u + v) \leq L(u) + L(v)$$

and the linear complexity of the sequence uv is

$$L(uv) \leq L(u)L(v).$$

If the two LFSRs were chosen such they are of relatively prime order, then the inequality will become an equality [4]. So in the previous example, if the order of $LFSR_1$ was 11, $LFSR_2$ was 7, $LFSR_3$ was 10, the total complexity of the combiner would be

$$f(L_1, L_2, L_3) = 11 + 7 * 10 = 81.$$

Another property of a Boolean function is whether or not it is balanced. A balanced Boolean function is defined in [4] as one that produces a even number of 0's and 1's in its output stream. Balanced Boolean functions are important because they are a prerequisite for having a random stream output. While not a guarantee of randomness, a non-balanced stream will not be random.

Example 2.2.1 shows the first 5 steps of the combiner from Equation 2.3.

Example 2.2.1 *Values from* $f(x_1, x_2, x_3) = x_1 \oplus x_2x_3$

In Table 2.2, the combiner takes the input from three different LFSRs. The combiner then acts on the input using the specified Boolean function to create an output bit.

Table 2.2. First 5 Steps of Equation 2.3

Step	$LFSR_1$	$LFSR_2$	$LFSR_3$	Output Bit
0	1	0	0	1
1	0	1	1	1
2	0	0	0	0
3	1	0	1	1
4	1	1	1	0
5	0	1	0	0

2.3 Generalized Boolean Functions

A generalized Boolean function is a function $f : \mathbb{Z}_{2^m} \rightarrow \mathbb{Z}_{2^n}$ of the form

$$f(x) = \sum_{i=0}^{k} 2^i a_k(x),$$

where a_k is in the set of feeding LFSR streams. A generalized Boolean function can take its input from regular Boolean functions. The difference between a generalized Boolean function and a regular Boolean function is that a generalized Boolean function outputs a stream in $\mathbb{Z}_{2^n}$ rather than a stream in $\mathbb{Z}_2$. For Example, taking $n = 1$ the form of the generalized Boolean function would be

$$f(a_0, a_1) = a_0 + 2a_1$$

and if $n = 2$ the form would be

$$f(a_0, a_1, a_2) = a_0 + 2a_1 + 4a_2.$$

Using a generalized Boolean function as a combiner is very similar to using a regular Boolean function as a combiner.

Not much is known about the complexity or randomness properties of generalized Boolean functions when they are implemented as a combiner. However, previous work has been done that shows that the regular Boolean combiner choices and their sum needs to be balanced. Finding a concrete way to calculate the linear complexity of the resultant stream is still an

open problem.

2.4 Complexity of Sequences

Linear complexity is an important topic because it is one indicator of how secure a sequence is. Definition 2.4.1 provides a definition of linear complexity.

Definition 2.4.1 *The linear complexity, L, is defined for a sequence, s, as the order of the shortest LFSR that generates s.*

For a primitive LFSR with degree L it is known that the linear complexity will also be L given a non-zero seed. Making primitiveness a requirement for LFSR selection is important because it maximizes the complexity of the LFSR. One way to calculate the complexity of a sequence is to use the Berlekamp-Massey Algorithm.

2.4.1 Berlekamp-Massey Algorithm

The Berlekamp-Massey Algorithm is an efficient way to calculate the linear complexity of a finite binary sequence. The algorithm works by calculating an LFSR that can generate the given finite sequence. Definition 2.4.2 and Algorithm 1 are taken from [3] to describe how and why the algorithm works.

Definition 2.4.2 *Consider the finite binary sequence* $s^{N+1} = s_0, s_1, \cdots, s_{N-1}, s_N$. *For* $C(D) = 1 + c_1 + \cdots + c_L D^L$, *let* $L, C(D)$ *be an LFSR that generates the subsequence* $s^N = s_0, s_1, \cdots, s_{N-1}$. *The next discrepancy* d_N *between* s_N *and the* $(N+1)^{st}$ *term generated by the LFSR is* $d_N = (s_N + \sum_{i=1}^{L} c_i s_{N-i}) \mod 2$.

Algorithm 1: Berlekamp-Massey Algorithm

Input: a binary sequence $s^n = s_0, s_1, s_2, \cdots, s_{n-1}$ of length n.

Output: the linear complexity $L(s^n)$ of s^n, $0 \leq L(s^n) \leq n$.

```
Initialization: C(D) = 1, L = 0, m = −1, B(D) = 1, N = 0;
while N < n do
    Compute the next discrepancy d;
    if d = 1 then
        T(D) = C(D);
        C(D) = C(D) + B(D) * D^{N−m};
        if L ≤ N/2 then
            L = N + 1 − L;
            m = N;
            B(D) = T(D);
        end
    end
    N = N + 1;
end
Return(L);
```

One of the important things to note about the algorithm is that it has complexity $O(n^2)$ [3]. As a result, the algorithm is efficient for calculating streams up to a certain length, but once the streams get long, it becomes very computationally intensive to run.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: Methodology

Through a combination of programs written by other researchers (for which we will give due credit) and programs we wrote, the data for this thesis was largely collected by generating multiple sequences and then testing the results. The first step of our data collection was to create a program that would generate our data. To accomplish this task we wrote a Python program (included in the Appendix) that output the results from multiple generalized Boolean functions to a file. We could then feed these sequences into our randomness tests and linear complexity programs. Additionally, steps had to be taken to convert the $\mathbb{Z}_n$ output stream from the generalized Boolean functions to a binary stream suitable for testing.

In total, three LFSRs were chosen that were primitive and of maximal period. The output of these LFSRs fed into seven sets of Boolean combiners. Of these seven sets, all were balanced and their sums were balanced. Five sets were of quadratic form, and two were of affine form. The output of the Boolean combiners was then used to generate the final stream from a choice of two different generalized Boolean functions used as combiners. The final stream was 10^6 bits long.

3.1 Randomness Tests

In order to test whether the stream produced through the generalized Boolean function was random, the 15 tests from the National Institute of Standards and Technology (NIST) statistical test suite were used. The tests were implemented using a Python program written by Ilja Gerhardt [5]. This program took as its input either a binary file with the output stream, or a text file with the output stream. We used text files to test our data because it ran with the least issues. These tests only work on binary streams so our $\mathbb{Z}_{2^n}$ output obtained from the generalized Boolean functions had to be converted into a binary sequence.

3.1.1 Converting to $\mathbb{Z}_2$

As mentioned in Section 2.3, the output of a generalized Boolean function is in $\mathbb{Z}_n$ where n is calculated by 2^{2^i}. In this thesis we used $i = 1$ so all of our output for the generalized

Boolean functions were in $\mathbb{Z}_4$. One of the requirements of the NIST randomness tests and the Berlekamp-Massey algorithms is that the tested sequence be a binary sequence. In order to use these tests, we had to apply the following transformation to our output streams

$$\begin{aligned} 0 &\rightarrow 00 \\ 1 &\rightarrow 01 \\ 2 &\rightarrow 10 \\ 3 &\rightarrow 11. \end{aligned}$$

We propose that this simple binary transformation is sufficient to not affect any of the randomness properties of the original stream. Care was taken to ensure that the result transformation was of a fixed length in order to ensure that each input integer had an equal value in the output stream. One of the consequences of this is that the output stream is twice as long as in the input stream. Additionally, this transformation ensures that there are no additional 0's or 1's inserted into the sequence that could potentially unbalance the resultant stream. While there are certain small sequences that are unbalanced or have repetition of bits as a result of the transformation, such as

$$201 \rightarrow 100001,$$

this is not of concern because when sufficiently large sequences are taken, these runs average out to not affect the overall randomness provided the original sequence was sufficiently random.

3.1.2 NIST Randomness Tests

In this section, we describe the 15 tests in detail and give a brief description of how to interpret their results. The tests and their implementation details are all thoroughly explained in the paper published by the NIST and the details about the tests come from [6]. The following section draws very heavily on their technical explanations. For the tests, a p-value of $p < 0.01$ is indicative of failing the randomness test.

The first test in the suite is the *Monobit Frequency test*. The *Monobit Frequency* test measures the proportions of ones and zeros in the entire sequence. In a truly random sequence, the number of zeros and ones should be in even proportion. This test calculates

whether the input stream is approximately in the same proportion as a truly random stream. If the p-value is very large, that indicates that their was either a disproportionally high number of zeros or ones in the sequence [6].

The second test is the *Block Frequency* test. Like the previous test, it measures the proportion of 0's to 1's. However, the difference is that rather than measure the ratio over the entire sequence, it measures it in size M-bit blocks. The proportion is calculated per block, with the expected frequency to be $M/2$ if the sequence was random. If the block size is $M = 1$, than it just the previous test. If sequence fails the test, this means that the proportion of ones to zeros is higher in at least one of the blocks measured [6].

The third test is the *Runs* test. A run can be defined as k matching bits in a row. For example, the sequence 01010001 has a run of 3 because there are 3 consecutive 0's. In a random sequence, there should not be any runs of exceptionally long length, and the runs should vary between either bit roughly an equal number of times. If 0's and 1's are equally likely, the probability of a run of length k is $1/2^k$. The first step of the runs test is to perform a frequency test. If the frequency test fails, then the runs test is not performed. The test statistic is given in [6] and can be calculated by

$$V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1,$$

where $r(k) = 0$ if $\epsilon_k = \epsilon_{k+1}$, and $r(k) = 1$ otherwise. If the value of $V_n(obs)$ is large, than the sequence switched runs too quickly. A small value means the stream switched slowly. This can be interpreted as a large value of $V_n(obs)$ meaning that there were too many short runs, and a small value of $V_n(obs)$ as being that there were too many long runs [6].

The fourth test is a test for the *Longest Runs of Ones* in a block. This test looks to see if the longest run of ones in an M-bit block is on par of the expected ones run of a random sequence. A large test statistic is indicative of large runs of ones [6].

The fifth test is the *Binary Rank Matrix* test. This test looks at the rank of disjoint sub-matrices of the entire sequence. NIST defines it as testing for "linear dependence among fixed length substrings of the original sequence" [6]. For this test, NIST has set the sub-matrices to be of size 32x32. The sub-matrices are filled in row by row from the original

sequence. The rank for each sub-matrix is then computed, and the test statistic calculated. A failing p-value would show that the rank distribution was not close to that of a random sequence [6].

The sixth test is the *Discrete Fourier Transform* or *Spectral* test. This tests looks at the peak heights of the Discrete Fourier Transform of the sequence. If the sequence is not random, then the transform would have groups of periodic features or patterns. The test aims to find "whether the number of peaks exceeding the 95% threshold is significantly different than 5%" [6]. The first step is to calculate the 95% peak threshold value, T, with

$$T = \sqrt{(}\log(\frac{1}{0.05})n).$$

The next step is to compute N_0, the number of theoretically expected 95% peaks by $N_0 = .95n/2$. Next, the actual number of peaks that exceed the 95% threshold, N_1 is tabulated. If the test fails, then the value of d was too low indicative of greater than 5% of peaks above the threshold [6].

The seventh and eighth tests are the *Non-overlapping Template* test and the *Overlapping Template* test. Both tests count the number of occurrences of pre-specified target strings in an m-bit window. This is to check to see if the sequence has a predictable patterns that would display a lack of randomness. The tests work by taking a m-bit window from the sequence, then comparing to the pre-selected string. For the non-overlapping template test, if there is a match with the window and string, then window advances to the next m-bit chunk. If there is not a match, the window advances 1 bit. For the overlapping template test, a match will cause the window to advance by only a bit. Additionally, a miss will cause the window to advance by only a bit. For both tests, NIST defines a small p-value as showing that "the sequence has irregular occurrences of the possible template patterns" [6].

The ninth test is *Maurer's "Universal Statistical"* test. This test measures whether the sequence can be successfully compressed without loss of information. According to the NIST guide, "a significantly compressible sequence is considered to be non-random" because patterns exist that the compression is able to take advantage of and reduce space [6]. This test looks at "the sum of $\log_2$ distances between matching L-bit templates" where L is the block size [6]. If a low p-value is obtained, this shows that the sequence is highly

compressible and not random.

The tenth test is the *Linear Complexity* test. Like stated in Section 2.4, the linear complexity of the sequence can be defined as the order of the minimum LFSR required to generate the sequence. A property of true random sequences is that have very high linear complexity. This test requires at least 10^6 bits in order to be valid. The first step of this test is to break the "n-bit sequence into N independent blocks of M bits, where n=MN" [6]. Then using the Berlekamp-Massey algorithm, calculate the linear complexity, L_i, of each of the blocks. Using the linear complexity of each block, calculations are performed using the theoretical mean in such a way to generate a chi squared distribution. If the p-value is below the randomness cutoff, then that would imply that the blocks were of insufficient linear complexity [6].

The eleventh test is the *Serial test*. One of the properties of a random sequence is uniformity. This translates to the fact that every m-bit sequence has equal probability of appearing as every other m-bit sequence. The test implements this by comparing the number of occurrences of 2^m m-bit overlapping patterns is similar to that of a random sequence. This test is performed by taking the sequence and appending $m-1$ bits to create an augmented sequence, then analyzing the frequency of all overlapping m-bit blocks. If the test fails, it shows that the m-bit blocks have non-uniformity [6].

The twelfth test is the *Approximate Entropy* test. This test is similar to the serial test in that both look at overlapping patterns in m-bit sequences. With the approximate entropy test, the area of interest is the frequency of m-bit sequences in two overlapping blocks offset by 1 bit. The blocks are generated the same way as the serial test, except that the second block is 1 bit larger. With each block, the number of overlapping m-bit sequences is tallied, then the test statistic is calculated. If the test fails because the test statistic is too small, then it implies "strong regularity" and if it is too large it implies "substantial fluctuation or irregularity" [6].

The thirteenth test is the *Cumulative Sums* test. This test aims to measure "whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences" [6]. If the sequence is random, then the expected result should be close to zero. For this test, the sequence has the 0's converted to -1, and the 1's kept as 1's. The next step is to calculate all

the partial sums up to the size of the sequence, then take the maximum value obtained. This test is conducted twice; once with the partial sums done from the front, and once with the partial sums calculated from the reversed sequence. These values are then used to calculate the p-value. If the value is too small for the first value, then there were too many 1's or 0's in the beginning of the sequence. If the second value is too small, then there were too many 1's or 0's in the end of the sequence [6].

The fourteenth and fifteenth tests are the *Random Excursions* and the *Random Excursions Variants* test. These tests deal with cumulative sums random walks and how often each sequence visits each state. Both test actually perform a series of individual tests which result in list of p-values and whose meaning is beyond the scope of this thesis [6].

3.2 Experimental Procedure

The Python program that generated our data contains 3 LFSRs, 7 sets of two combiners and 2 generalized Boolean functions. The output of the LFSRs was used to feed all 7 sets of combiners. For every set of combiners, the same two generalized Boolean functions were applied. This was done to try and discern an effect of combiner choice.

3.2.1 LFSRs Choice

The LFSRs that were chosen for use are listed in Table 3.1.

Table 3.1. The 3 Choices of LFSRs

LFSR	Polynomial Form
$LFSR_1$:	$x^{11} + x^6 + x^5 + x^4 + x^3 + x + 1$
$LFSR_2$:	$x^{10} + x^3 + 1$
$LFSR_3$:	$x^7 + x^5 + x^3 + x + 1$

The LFSRs were seeded using the Python rand() functions, which is (somewhat) cryptographically secure PRNG and suitable for our applications. The order of the three LFSRs were 7, 10, and 11. The choice of these three LFSRs made sure that order of all three

were co-prime for reasons explained in Section 2.1. Additionally, all three are primitive polynomials. The output of the LFSRs was fed into the choice of Boolean combiners.

3.2.2 Boolean Combiner Choice

The combiners were chosen to highlight the two different situations of having a quadratic choice and an affine choice. The first five sets of combiners were pairs of quadratic Boolean functions. The last two sets of combiners were pairs of affine Boolean functions. The 5 sets of quadratic combiners are listed in Table 3.2.

Table 3.2. The 5 Choices of Quadratic Combiners Where x_i Corresponds to Output Bit of $LFSR_i$

Set		Combiner Choices
Set 1	C_1:	$x_0x_1 + x_0x_2 + x_2$
	C_2:	$x_0x_1 + x_2$
Set 2	C_1:	$x_0x_1 + x_0x_2 + x_1$
	C_2:	$x_0x_2 + x_2x_1 + x_1$
Set 3	C_1:	$x_0x_2 + x_1 + x_2$
	C_2:	$x_0x_1 + x_1x_2 + x_2$
Set 4	C_1:	$x_1x_2 + x_0x_2 + x_0$
	C_2:	$x_1x_2 + x_0$
Set 5	C_1:	$x_1x_2 + x_0x_1 + x_0$
	C_2:	$x_0x_2 + x_1$

The 2 sets of affine Boolean functions are listed in Table 3.3.

Table 3.3. The 2 Choices of Affine Combiners Where x_i Corresponds to Output Bit of $LFSR_i$

Set		Combiner Choices
Set 6	C_1:	$x_0 + x_1$
	C_2:	$x_1 + x_2 + 1$
Set 7	C_1:	$x_0 + x_2 + 1$
	C_2:	$x_1 + x_2$

Both affine and quadratic Boolean functions were used to explore the effect of combiner

choice on the generalized Boolean function, since it is unknown what effect the order of the Boolean combiner choice has on the output of the generalized Boolean function stream. All the combiners chosen are balanced, and the sum of each combiner in a set is also balanced. The reason that it is important that the sum is balanced is based off the work of [7] which proved that some good cryptographic properties for the generalized Boolean functions are obtained if linear combinations of its components are balanced. We extend this notion by claiming that imposing a balanced sum on its components will cause better behavior on the generalized Boolean function output streams.

3.2.3 Generalized Boolean Function Choice

As mentioned previously, a generalized Boolean function outputs its stream in $\mathbb{Z}_n$. We took our generalized Boolean functions to be in $\mathbb{Z}_4$. Table 3.4 lists the equations that were used for our generalized Boolean functions.

Table 3.4. The 2 Choices of Generalized Boolean Functions Where a_i Corresponds to Output Bit of C_i

GBF Choice	
gf_1:	$x_0 + 2x_1$
gf_2:	$x_1 + 2x_0$

Both gf_1 and gf_2 were applied against every set of combiners from Table 3.2 and Table 3.3. We took the lengths of the final stream to be 10^6 bits because that was the minimum number of bits required for some of the randomness tests. The generalized Boolean functions were then written to a text file for analysis.

3.3 Testing Sequences

After the streams were written to the file, two batteries of tests were performed. The first test was a test of linear complexity. This was accomplished using the Berlekamp-Massey Algorithm. The program that was used returned the order of the minimum generating LFSR for that sequence. This test was performed on every generalized Boolean function combiner stream. This test required that the output text file contains spaces between every bit.

The second test was the NIST randomness test battery. The NIST test's output the p-value results from each test. The randomness tests were run against the first choice of the generalized Boolean functions generated from set 1 and set 6 for a stream of 10^6 bits in length. This test required that the output text file contain no spaces between every bit.

In addition, we looked at the frequency of each integer of the generalized Boolean function stream to see if any patterns were apparent.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results and Analysis

Results from the linear complexity and randomness tests, in addition to some analysis on this data will be provided in this chapter. One of the major results of this thesis was that significant increases in linear complexity were obtained by using the generalized Boolean function as a combiner of the Boolean combiner input. Additionally, our data shows that high degrees of randomness can be achieved using either the higher order sets or the affine sets.

As mentioned in Section 2.2 and 3.2.2, we hypothesized from the work of [7] that imposing the requirement of having the sum of the combiners be balanced was necessary for achieving a usable result. During preliminary testing of our data, we tried using multiple sets of combiners who sum was not balanced, and the results were very poor. Since the effect was so obvious, this thesis will only report the data from sets of combiners who sum is balanced.

4.1 Linear Complexity Results

The increase in linear complexity obtained from using a generalized Boolean function as a combiner is one of the most significant results of this thesis. In Table 4.1, the linear complexity of each combiner of every set and the resultant linear complexity of the generalized functions is listed. Additionally, the average complexity of the Boolean combiners per set is given.

From this data several things are apparent. The first thing of notice is how the choice of generalized Boolean function does not affect the linear complexity of the output stream. This can be explained simply using the structure of the generalized Boolean function. Recalling Section 2.3 and Section 3.2.3, the structure of the two generalized Boolean functions we used was

$$f(a_0, a_1) = a_0 + 2a_1.$$

This function will output a stream in $\mathbb{Z}_4$. To generate each of elements certain values of a_0 and a_1 need to be used. To generate a 0, $a_0 = 0$ and $a_1 = 0$. To generate a 1, $a_0 = 1$ and $a_1 = 1$. This continues until the entire set is generated. Looking at these facts, we see that

Table 4.1. The Linear Complexity of the Generated Streams

Set 1				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
158	81	119.5	550	550

Set 2				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
157	197	534	534	177

Set 3				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
98	191	548	548	144.5

Set 4				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
194	117	316	316	155.5

Set 5				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
187	87	556	556	137

Set 6				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
17	22	58	58	19.5

Set 7				
$L(C_1)$	$L(C_2)$	$L(gf_1)$	$L(gf_2)$	$(L(C_2)+L(C_1))/2$
18	21	58	58	19.5

the only thing that changes between the output of gf_1 and gf_2 is when a 1 is generated in gf_1, a 2 is generated in gf_2. Similarly, when a 2 is generated in gf_1, a 1 is generated in gf_2. Table 4.2 shows the frequency of occurrences from the data generated by Set 1.

Looking at this data, the number of 1's and 2's generated in gf_1 matches the number of 2's and 1's generated in gf_2 while the numbers of 0's and 3's does not change. We propose

Table 4.2. Frequency of Integers of Stream from Set 1

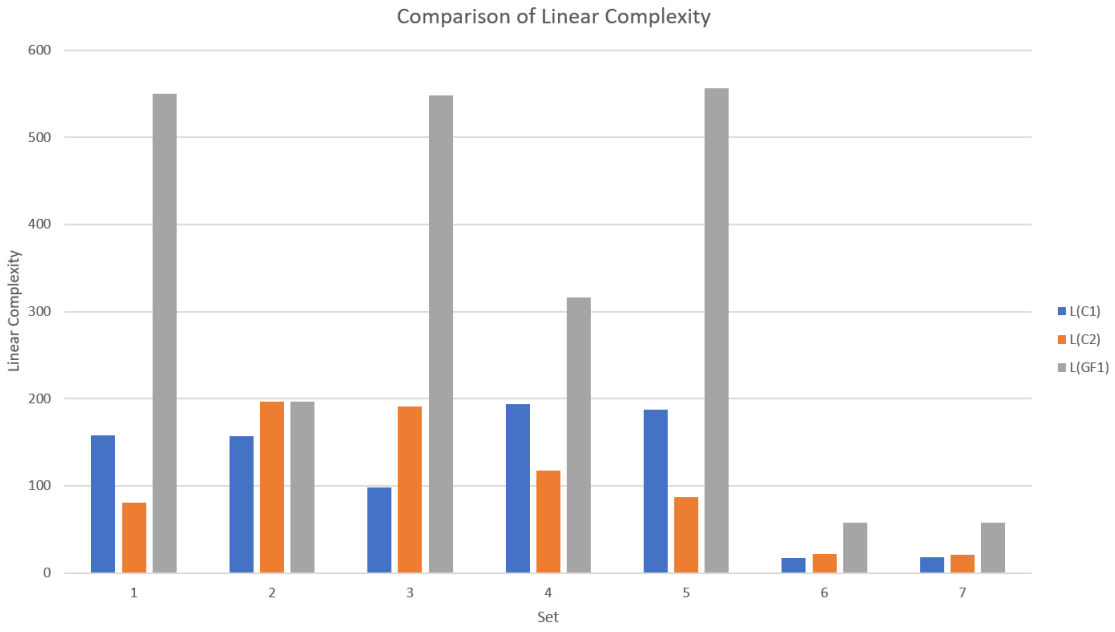
gf_1							
0:	124040	1:	126076	2:	124863	3:	125021

gf_2							
0:	124040	1:	124863	2:	126076	3:	125021

that this explains why the linear complexity of both generalized functions is the same. This lack of change is the result of a simple transformation that has no effect on the complexity of the generating stream because it is just simply switching in place every 1 and 2.

Another interesting result is the significant increase in complexity of the gf_1 and gf_2 streams from the regular Boolean combiners. Figure 4.1 provides a chart that shows the large increases.

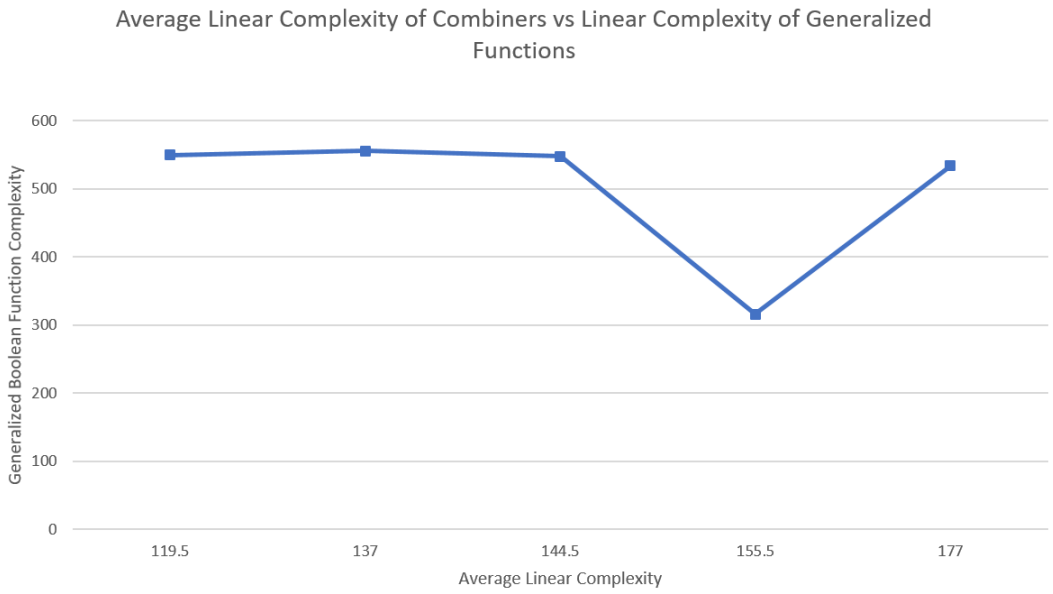
Figure 4.1. Linear Complexity of the Generalized Boolean Functions and Their Input Combiners



From this it is clear the significant increases in complexity that is occurring. In almost all sets, roughly a three-fold increase in complexity is achieved over the average input complexity. Even in the affine sets, the increase is significant. However, in Set 7, the complexity of the generated stream is less than then the others. We have not found a satisfactory reason to fully explain this result, however since all of the Sets used the same LFSR data and applied the same generalized Boolean functions, we suspect this is a result of an unknown interaction between the combiners that were chosen.

In Figure 4.2, the average linear complexity of the Boolean combiners of each quadratic set is plotted against the linear complexity of the generalized Boolean function.

Figure 4.2. The Average Complexity of a Input Set Versus the Complexity of the Generalized Boolean Function



One of the interesting things this graph shows is the general lack of an upward trend in complexity as average linear complexity increased. Intuitively, it would make sense that as average linear complexity increased, the average generalized Boolean function complexity would also increase, resulting in an line of slope 1 in Figure 4.2. However, this data seems to show a lack of an increase as the average input complexity increases. In fact, the value

of Set 4 of 316 does not fit into the general pattern at all. This data would seem to suggest that the complexity of the generalized Boolean function stream is not directly dependent on the complexities of its inputs. However, this data set only has 5 points, and is therefore not sufficiently large to warrant drawing any wide reaching conclusion.

4.2 Quadratic Set Randomness Results

This section will present the data from the randomness tests from Section 3.1 which were run against our data. For the tests, the Serial test was not included since there was an error in the code that prevent it from completing successfully. Completion of the randomness tests took a significant amount of time due to the length of the streams used. We therefore looked specifically at the results from Set 1 and Set 6 for this section.

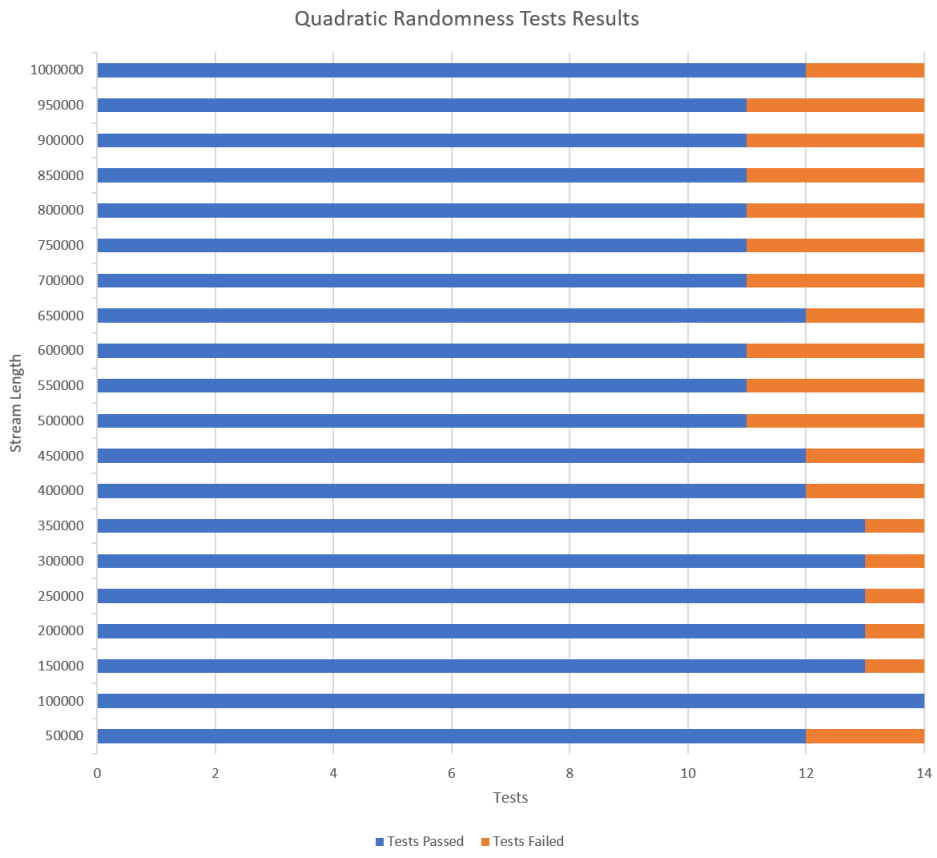
Figure 4.3 shows the number of NIST tests of various stream lengths which passed. A p-value of less than 0.01 is considered failing. For this set of tests, we looked at the effect of increasing the stream length on randomness. Since our generalized Boolean function have linear complexities that are around 500, we suspected that as the stream length approaches a million the performance on sum of the tests decreases due to the relatively smaller linear complexity of the sequence.

Looking these results, there does appear to be a slight decrease in randomness performance as stream length increases. Overall though, passing a minimum of 11 tests is still a good result. The stream length of 100,000 passed all the tests, which is a very good result. Figure 4.4 shows the p-values of the individual tests, the decreasing trend is more apparent here.

From this figure it is clear that there is a general downward trend as the stream length increases. Since the minimum recommended length for several of the NIST tests is one million bits, Figure 4.5 displays the p-values from the results of the one million bit stream.

Looking at specific tests, some of the results are expected. For example, the failing linear complexity p-value is to be expected since the linearity of the sequence is so low relative to stream length. Additionally, the tests that focus on blocks, like the *Block Frequency Test*, *non-Overlapping* and *Overlapping tests*, have higher p-values since they do not look at the entire stream length at a time, so having a longer stream will not hurt that score.

Figure 4.3. Number of Randomness Tests the Quadratic Set Passed by Stream Length

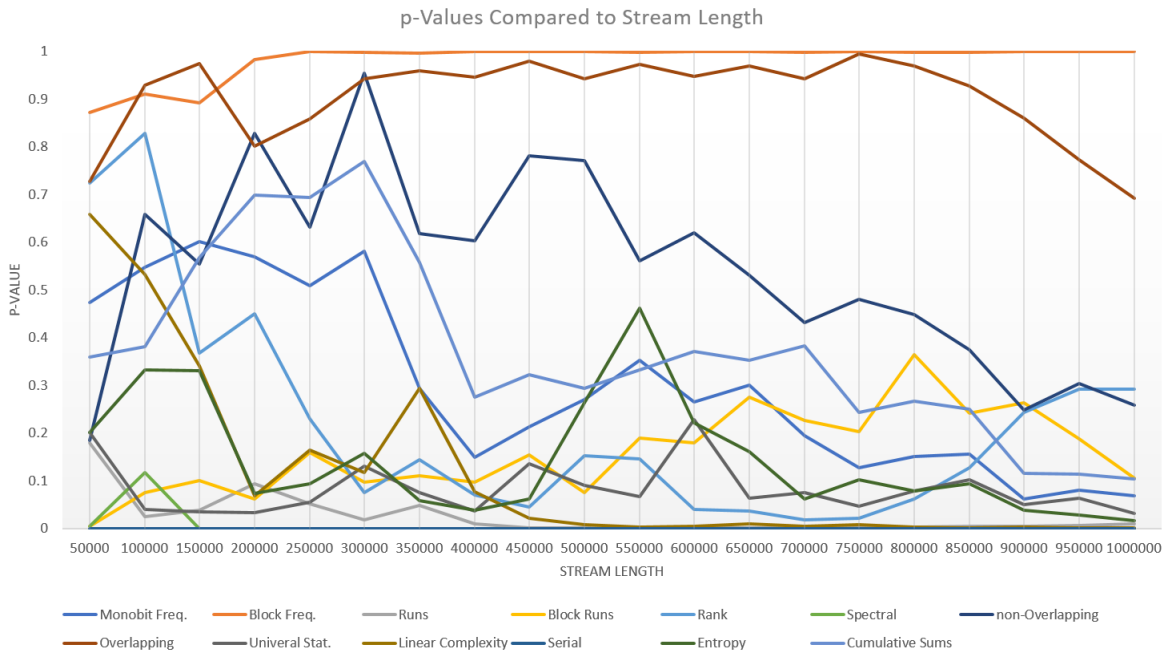


Overall for the quadratic tests, the results are logical and show that good levels of randomness can be obtained using the generalized Boolean function as a combiner.

4.3 Affine Set Randomness Results

This section will present the randomness results of affine sets. We decided to test the affine sets because we thought that the linear nature of the affine functions would create poor randomness. Our results however seemed to suggest that the affine streams had good levels of randomness. Figure 4.6 shows the number of passed tests for the affine set as stream length increases.

Figure 4.4. Effect on p-value as Stream Size Increases



With passing an average of 11 tests, the affine results seem to suggest that this is a good source of randomness. However, this set is lacking the downward trend that was seen in the quadratic set. In Figure 4.7, this lack of trend is apparent.

Looking at this data it shows that the p-values remain roughly consistent over stream length. Given all these facts, and comparing to the results of the quadratic functions we are hesitant to draw a conclusion on the randomness of the affine sets.

It is worth noting, that for both the quadratic and affine sets, we generated a new sequence of data to see if the results were consistent across many trails and found a high degree of variability. This could be due to the small linear complexity of the streams, but its exact reason is unknown. Additionally, this variability was only for the randomness tests and not the linear complexity, so that may indicate it was a problem with the program used for testing and not the data.

Figure 4.5. p-values for Quadratic Set of One Million Bits by Test

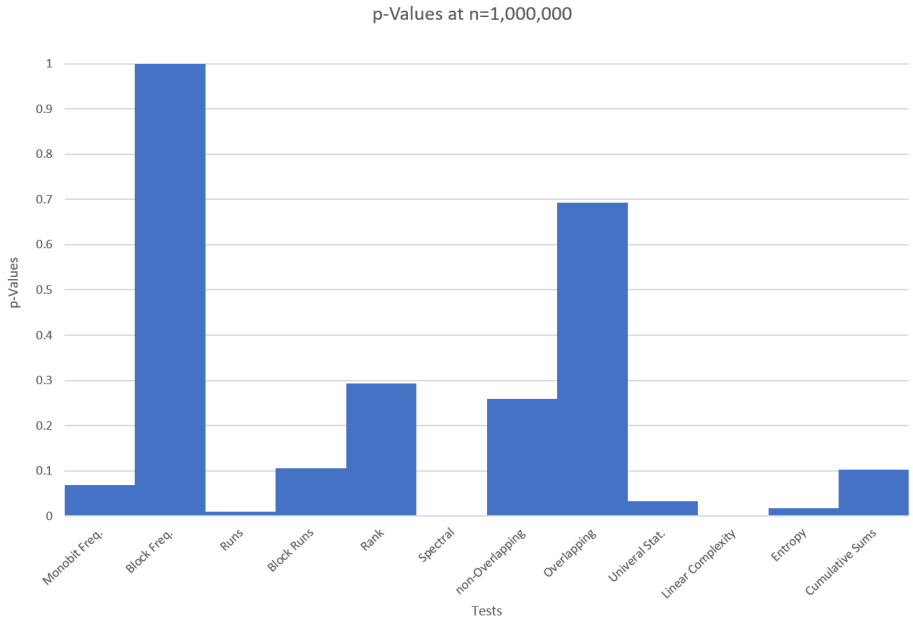


Figure 4.6. Number of Randomness Tests the Affine Set Passed

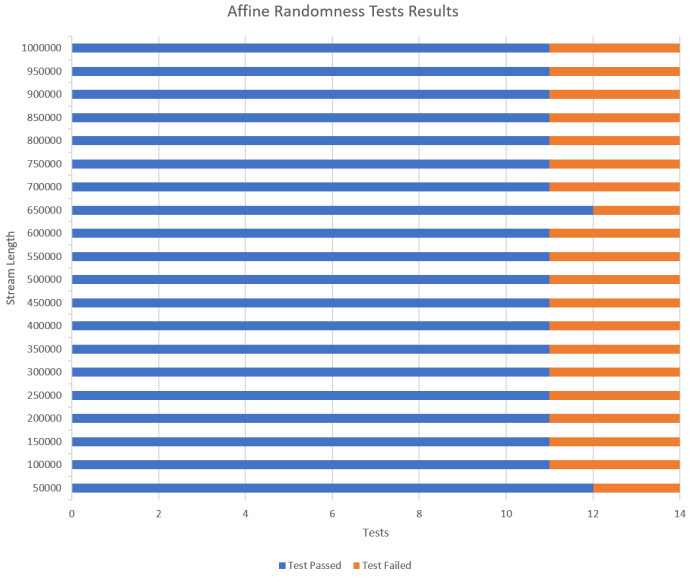
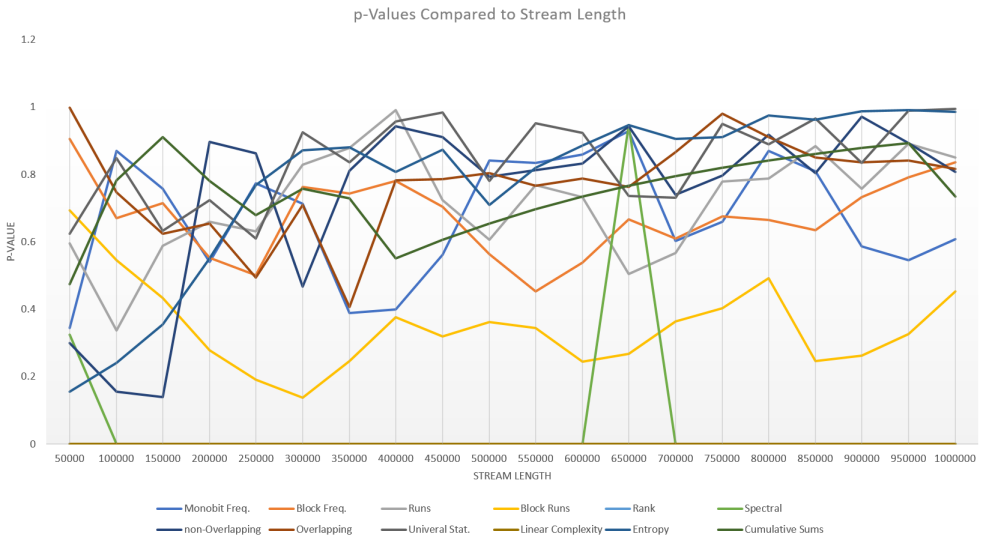


Figure 4.7. Effect on p-value as Stream Size Increases



THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

5.1 Conclusion

This thesis proposes a PRNG using generalized Boolean functions as combiners and has shown the streams exhibit increases in complexity as well as appear to be random. The most significant take away from this thesis is the large increase in linear complexity. However, we are unable to find an analytically way of calculating a number for the complexity. Given the data we presented, there is not a obvious way to predict what the complexity will be. This could be a opportunity for future work in this area. As is shown in Figure 4.2, the complexity does not appear to be directly dependent on average input size. We suspect that it had more to do with the choice of combiners and some interaction that is occurring. Additionally, our data set was small, so any large patterns might not be seen. For future work, we suggest taking LFSRs of significant higher complexity and taking the generalized Boolean function to be in a larger space so that more varied data, and data that mimics what might be actually used in application, can be gathered. Also further study exploring the interaction between the combining Boolean functions could be fruitful. We want to stress that while the tests we performed only tested Boolean combiners that were of affine and quadratic degrees, we do not think that the results only apply to those degrees. Future work with combiners of higher degree could confirm this, but there is no reason to limit these results to only quadratic combiners. Overall, these results are promising and significant and are a good start for an area where not much is known.

For the randomness results of the streams, we found that the quadratic sets of the combiner choices exhibit good randomness properties. We found that they tended to decrease as size of bit stream increased, which was expected. For future work we think that increasing the complexity of the sequence would help get more randomness results because it would allow the tests to have the streams to be more suitable for the one million bit stream length required for some of the tests. For the affine tests, we found that they reported good results, however we were hesitant to draw conclusions about the data due to the very linear nature of the data. Since affine functions are never used in application, this result is not significant. For

the randomness tests, future work should also aim to repeat these results as we had some variability in our data in successive trials.

One thing that would be interesting for future work is to try and rewrite some of the NIST tests to work natively in integers. While we do not think our transformation to a binary stream hurt our results, it would be interesting if there was a measurable difference. Additionally, this thesis intentionally did not mention any cryptanalysis attacks that the generalized Boolean function might be susceptible to. Another fruitful endeavor would be to explore the vulnerably of these generators to common PRNG attacks.

In conclusion, this thesis explored an area where not much is known. We found some very promising and interesting data on the significant increase in linear complexity that comes from using a generalized Boolean function, and found that the streams exhibit good randomness properties. In the future, much work can be done expanding what we have accomplished.

APPENDIX: Code Used for Generating Streams

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Code Used for Generating Streams

A.1 Python Code for Generating Streams

The following is a listing of the Python code that was used to generate the data used in this thesis. It was designed to run on Python 2.7 and is operating system independent.

```
1  #!/usr/bin/env python
2
3  import random
4
5  #AUTHOR
6  #Oliver Di Nallo
7  #2LT, US Army
8  #Program Written for Masters Thesis "Generalized Boolean
       Functions as Combiners"
9
10 #USAGE:
11 #To use, run program. Generalized Boolean function combiner
       output will be written to file specified in output stream
12 #Adjust intStream to specify output, considering binary
       transformation will double stream length
13 #written for Python 2.7
14
15
16 INITIALIZATION
17 # each lfsr is represented by an array of ints that correspond to
        the taps
18 # lfsr1 = x^11 + x^6 + x^5 + x^4 + x^3 +x^1 + 1          # bits =
       11
19 lfsr1Tap = [11,6,5,4,3,1]
20 # lfsr2 = x^10 + x^3 + 1
                                  # bits = 10
```

```
21 lfsr2Tap = [10,3]
22 #lfsr3 = x^7 + x^5 + x^3 + x + 1
             # bits = 7
23 lfsr3Tap = [7,5,3,1]
24 taps = [lfsr1Tap, lfsr2Tap, lfsr3Tap]
25
26 #finds the degree of lfsr
27 maxTap1 = max(lfsr1Tap)
28 maxTap2 = max(lfsr2Tap)
29 maxTap3 = max(lfsr3Tap)
30 maxTaps = [maxTap1, maxTap2, maxTap3]
31
32 #get random seed for lfsrs
33 lfsr1 = 0
34 lfsr2 = 0
35 lfsr3 = 0
36 #ensures non-zero intial seed
37 while (lfsr1 == 0) | (lfsr2 == 0) | (lfsr3 == 0):
38         lfsr1 = random.getrandbits(11)
39         lfsr2 = random.getrandbits(10)
40         lfsr3 = random.getrandbits(7)
41 lfsrs = [lfsr1, lfsr2, lfsr3]
42
43 #initialize arrays for storage of lfsr, combiner, and generalized
      function output
44 #if increasing or decreasing number of any equations uses, be
     sure to update array sizes accordingly
45 lfsrBit = [0,0,0]
46 combinerBit = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
47 genFuncBit = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
48 genFuncString = ["", "", "","","","","","","","","","","","","","","
49 genFuncStringSpace = ["", "", "","","","","","","","","","","","","","
     ]
50
51 #set this to spefic y output stream
```

```
52 intStreamLength = 500000
53 j=0
54
55
56 while j<intStreamLength:
57         #LFSRS
58         for i in range(0,len(lfsrs)):
59                 xi = reduce(lambda x, y: x^y, map(lambda x: lfsrs
                       [i] >> (maxTaps[i]-x), taps[i]))&1
60                 lfsrBit[i] = lfsrs[i]&1
61                 lfsrs[i] = (lfsrs[i] >> 1) | (xi << (maxTaps[i
                       ]-1))
62
63         #COMBINERS
64         #see thesis for algebraic normal form of combiners
65         #quadratic combiners set 1
66         combinerBit[0] = lfsrBit[0] & lfsrBit[1] ^ lfsrBit[0]&
              lfsrBit[2] ^ lfsrBit[2]
67         combinerBit[1] = lfsrBit[0] ^ lfsrBit[1] & lfsrBit[2]
68         #quadratic combiners set 2
69         combinerBit[2] = lfsrBit[0]&lfsrBit[1] ^ lfsrBit[0]&
              lfsrBit[2] ^ lfsrBit[1]
70         combinerBit[3] = lfsrBit[0]&lfsrBit[2] ^ lfsrBit[2]&
              lfsrBit[1] ^ lfsrBit[1]
71         # #quadratic combiners set 3
72         combinerBit[4] = lfsrBit[0]&lfsrBit[2] ^ lfsrBit[1] ^
              lfsrBit[2]
73         combinerBit[5] = lfsrBit[0]&lfsrBit[1] ^ lfsrBit[1]&
              lfsrBit[2] ^ lfsrBit[2]
74         #quadratic combiners set 4
75         combinerBit[6] = lfsrBit[1]&lfsrBit[2] ^ lfsrBit[0]&
              lfsrBit[2] ^ lfsrBit[0]
76         combinerBit[7] = lfsrBit[1]&lfsrBit[2] ^ lfsrBit[0]
77         #quadratic combiners set 5
```

```
78          combinerBit[8] = lfsrBit[1]&lfsrBit[2] ^ lfsrBit[0]&
                lfsrBit[1] ^ lfsrBit[0]
79          combinerBit[9] = lfsrBit[0]&lfsrBit[2] ^ lfsrBit[1]
80          #affine combiners set 6
81          combinerBit[10] = lfsrBit[0] ^ lfsrBit[1]
82          combinerBit[11] = lfsrBit[1] ^ lfsrBit[2] ^ 1
83          #affine combiners set 7
84          combinerBit[12] = lfsrBit[0] ^ lfsrBit[2] ^ 1
85          combinerBit[13] = lfsrBit[1] ^ lfsrBit[2]
86
87          #Generalized Boolean Functions
88          #its the same two functions applied to each set of
                combiners
89          #reference thesis for algebraic normal form
90          #quad set 1
91          #gf1
92          genFuncBit[0] = (combinerBit[0] + 2*combinerBit[1]) % 4
93          #gf2
94          genFuncBit[1] = (combinerBit[1] + 2*combinerBit[0]) % 4
95          #quad set 2
96          #gf1
97          genFuncBit[2] = (combinerBit[2] + 2*combinerBit[3]) % 4
98          #gf2
99          genFuncBit[3] = (combinerBit[3] + 2*combinerBit[2]) % 4
100         #quad set 3
101         #gf1
102         genFuncBit[4] = (combinerBit[4] + 2*combinerBit[5]) % 4
103         #gf2
104         genFuncBit[5] = (combinerBit[5] + 2*combinerBit[4]) % 4
105         #quad set 4
106         #gf1
107         genFuncBit[6] = (combinerBit[6] + 2*combinerBit[7]) % 4
108         #gf2
109         genFuncBit[7] = (combinerBit[7] + 2*combinerBit[6]) % 4
110         #quad set 5
```

```
111         #gf1
112         genFuncBit[8] = (combinerBit[8] + 2*combinerBit[9]) % 4
113         #gf2
114         genFuncBit[9] = (combinerBit[9] + 2*combinerBit[8]) % 4
115         #affine set 1
116         #gf1
117         genFuncBit[10] = (combinerBit[10] + 2*combinerBit[11]) %
                4
118         #gf2
119         genFuncBit[11] = (combinerBit[11] + 2*combinerBit[10]) %
                4
120         #affine set 2
121         #gf1
122         genFuncBit[12] = (combinerBit[12] + 2*combinerBit[13]) %
                4
123         #gf2
124         genFuncBit[13] = (combinerBit[13] + 2*combinerBit[12]) %
                4
125
126         for i in range(0,len(genFuncBit)):
127                 binString = bin(genFuncBit[i])[2::].rjust(2,'0')
128                 string = binString[0]+binString[1]
129                 space  = binString[0]+"␣"+binString[1]+"␣"
130                 genFuncString[i]+=string
131                 genFuncStringSpace[i]+=space
132
133         j+=1
134
135 fileNames = ['quad1_gf1','quad1_gf2','quad2_gf1','quad2_gf2','
        quad3_gf1','quad3_gf2','quad4_gf1','quad4_gf2','quad5_gf1','
        quad5_gf2','affine1_gf1','affine1_gf2','affine2_gf1','
        affine2_gf2']
136
137
```

```
138 #uncomment this block if you want the output written as binary
        data
139 # for j in range(0,len(fileNames)):
140 #         fileName = fileNames[j]+".bin"
141 #         with open(fileName, "wb") as g:
142 #                 g.write(genFuncString[j].decode('hex'))
143
144
145 # #uncomment this block if you want the output written as a text
        file with NO spaces
146 # for t in range(0,len(fileNames)):
147 #         fileName = fileNames[t]+".txt"
148 #         with open(fileName, "w") as q:
149 #                 q.write(genFuncString[t])
150
151 # #uncomment this block if you want the output written as a text
        file with spaces
152 # for t in range(0,len(fileNames)):
153 #         fileName = fileNames[t]+".txt"
154 #         with open(fileName, "w") as q:
155 #                 q.write(genFuncStringSpace[t])
```

List of References

[1] Information Assurance Directorate. (2015, Aug). Commercial national security algorithm suite. [Online]. Available: https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm

[2] M. Haahr. True random number service. [Online]. Available: https://www.random.org

[3] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 2001.

[4] A. Canteaut, *Encyclopedia of Cryptography and Security*, H. C. A. Van Tilborg and S. Jajodia, Eds. Boston, MA: Springer US, 2011.

[5] I. Gerhardt. Random number testing. [Online]. Available: https://gerhardt.ch/random.php

[6] A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, A. Heckert, J. Dray, and S. Vo, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," *NIST Special Publication*, vol. 800-2, Apr 2010.

[7] T. Martinsen, W. Meidl, and P. Stănică, *Generalized Bent Functions and Their Gray Images*. Cham: Springer International Publishing, 2016, pp. 160–173. Available: http://dx.doi.org/10.1007/978-3-319-55227-9_12

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California